6. Queues

- Motivation
- Queue Operations
- Queue Interface
- Array Based Queues
- Circular Queues
- Pointer Based Queues
- Checking for Palindromes
- Queue Based Flood Fill
- Discrete Event Simulations
- Other Queue Applications
- Process Scheduling
- Queue Discussion

Motivation

- Queues were invented as an ADT for "last in last out" data storage.

- Queues are used to provide "fair service" to people in banks, buying tickets, etc.

- Queues are also used in many computer systems to provide "fair services".
  - printer queues
  - operating system scheduling queues
  - communication buffers
  - event simulation

Queue Operations

- The Queue ADT normally has the following operations.

  create - make a new queue

  destroy - delete all data on queue

  insert - put data at the end of the queue

  remove - get data from the front of the queue

  full - check for more room in the queue

  empty - checks queue for any data available

- Insert and Remove are also called Enqueue and Dequeue

Queue Interface

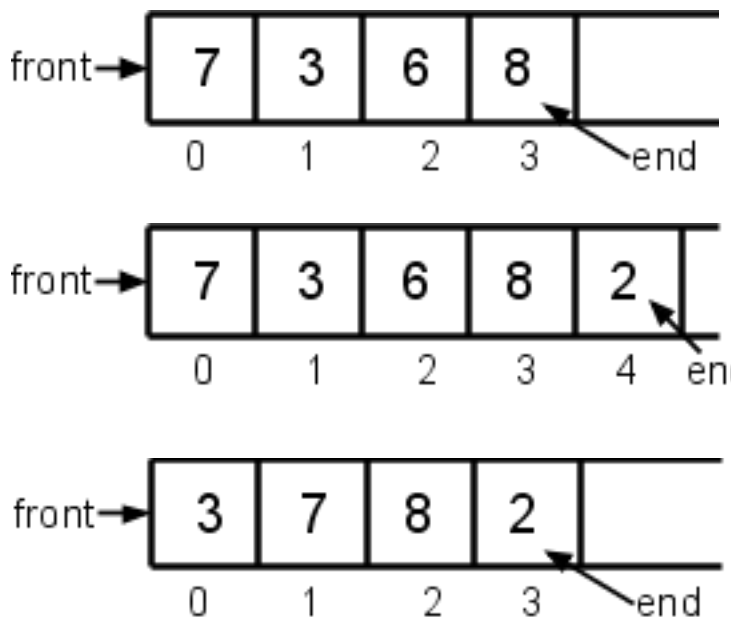
- Assume we want to store integers in a Queue.

```
class Queue
{
   public:
      Queue();
      ~Queue();

      void insert(int  item);
      int remove();

      bool isFull();
      bool isEmpty();

   private:
       TBA;
}
```

- We can change **int** and store any other data type in the queue.

Array Based Queues

- We can implement a queue using a fixed size array and an integer "end" that keeps track of where last data was inserted.
- By default, we assume that the "front" is in position 0 and move data in queue accordingly.



- initial queue
- insert will increment "end" and put data in that position
- remove will give us data at index 0 and shift all data left and decrement "end"

- This approach requires a lot of data to be copied.

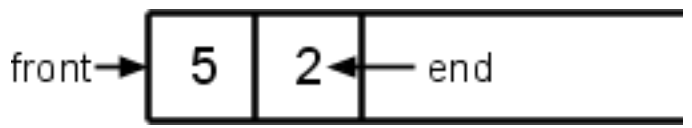
Array Based Queues

```
void insert( int item )
{
    //check size
    if( end < MAX_SIZE )
        data[++end] = item;

    //print error message
    else
        cout << "queue overflow\n"
}
```

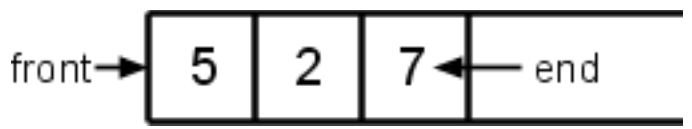
```
int remove()
{
    int item = data[0];

    //shift data
    for(int i=0; i<end; i++)
        data[i] = data[i+1];

    //return result
    if( end >= 0 )
        data[end--] = -1;

    return item;
}
```

Circular Queues

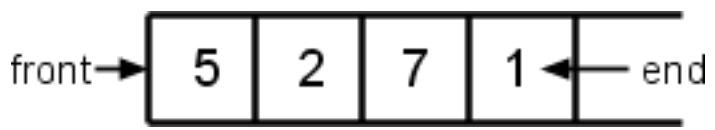
- We can improve on the simple array based queue by moving "front" and "end" to the right as data is inserted or removed.



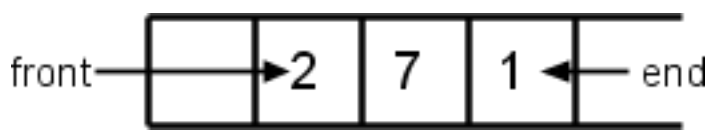
- initial queue



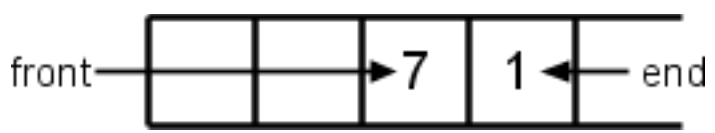
- after insert(7)



- after insert(1)



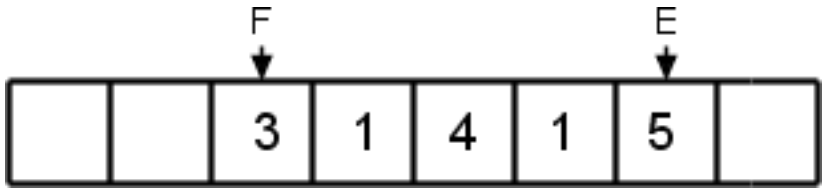
- after remove()



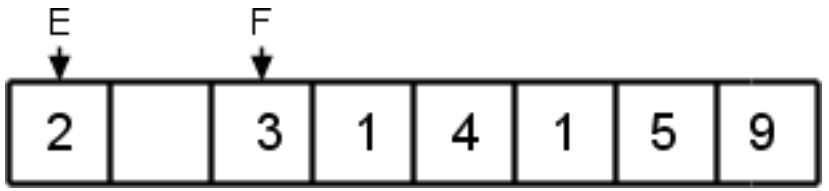
- after 2nd remove()

- We avoid array bounds error by wrapping the "front" and "end" indices around.

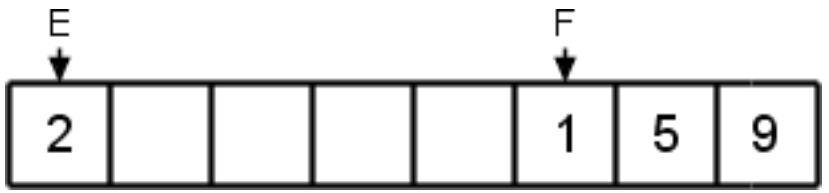
Circular Queues



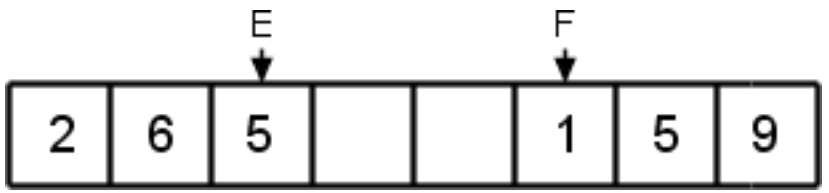
- initial queue



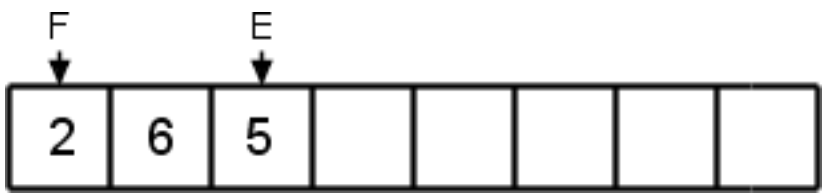
- after 2 inserts



- after 3 removes



- after 2 inserts



- after 3 removes

- In both cases we use modulo to wrap indices "index=(index+1)%size".
- Must also be careful to specify what empty and full look like. (so we can tell them apart).
- One solution is to add a counter to keep track of the number of items in the queue.

Circular Queues

```
void insert( int item )
{
   //check size
   if( count < MAX_SIZE)
   {
      end=(end+1)%MAX_SIZE;
      data[end]=item;
      count++;
   }
}
```

```
int remove()
{
   //check size
   if( count > 0 )
   {
      int item = data[front];
      front=(front+1)%MAX_SIZE;
      count--;
      return item;
   }
   return -1;
}
```

- Code above fails "quietly".

Circular Queues

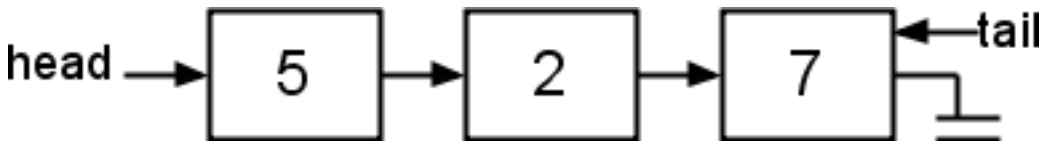
```
queue()
{
   count = 0;
   end = -1;
   frount = 0;
}
```

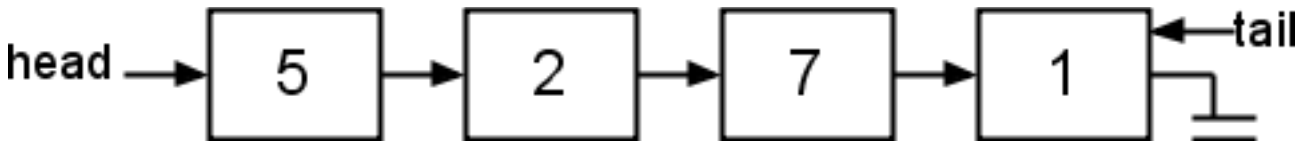
- Assume fixed size data array is defined in class.

```
bool isFull()
{
   return ( count >= MAX_SIZE );
}

bool isEmpty()
{
   return ( count <= 0 );
}
```

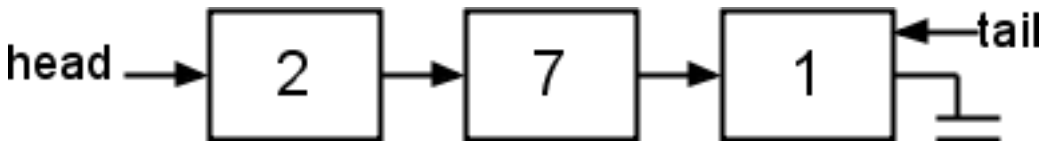
Pointer Based Queues

- We create a dynamic queue using linked list operations and inserting at the tail and removing from the head, or vice versa.





- after insert



- after remove

- Queue cannot be full unless we run out of memory
- No longer need to worry about wrap around code.
- Slightly slower than array based version.
- Which takes less space?

Pointer Based Queues

```
void insert( int item )
{
   queueList.insertTail(item);
}

int remove()
{
   //check empty queue
   if( !queueList.isEmpty())
   {
      int item=queueList.removeHead();
      return item;
   }
   //handle underflow
   else
      return -1;
}
```

- Another option would be to cut and paste the code for insertTail() and removeHead() into these functions.

Checking for Palindromes

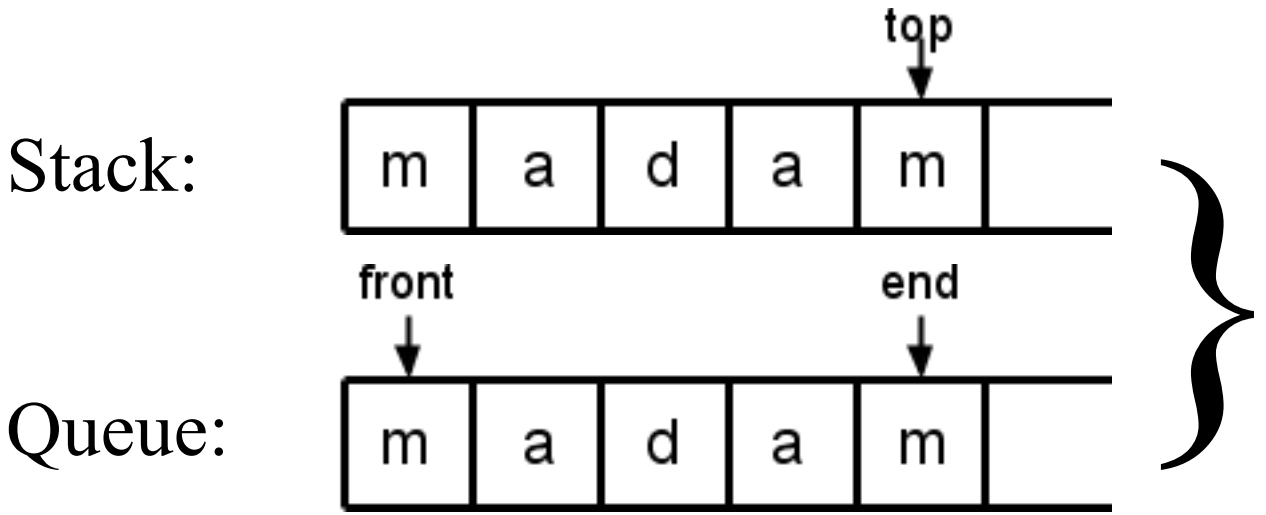
- Assume you are reading an unknown number of input characters.
- How do you check to see if it forms a palindrome?
- One solution is to push input characters onto a stack and insert them into a queue.
- When EOF is reached, we pop from the stack and remove from the queue until both are empty.
- If all characters from the stack and queue match each other as they are popped/removed, its a palindrome.

Checking for Palindromes

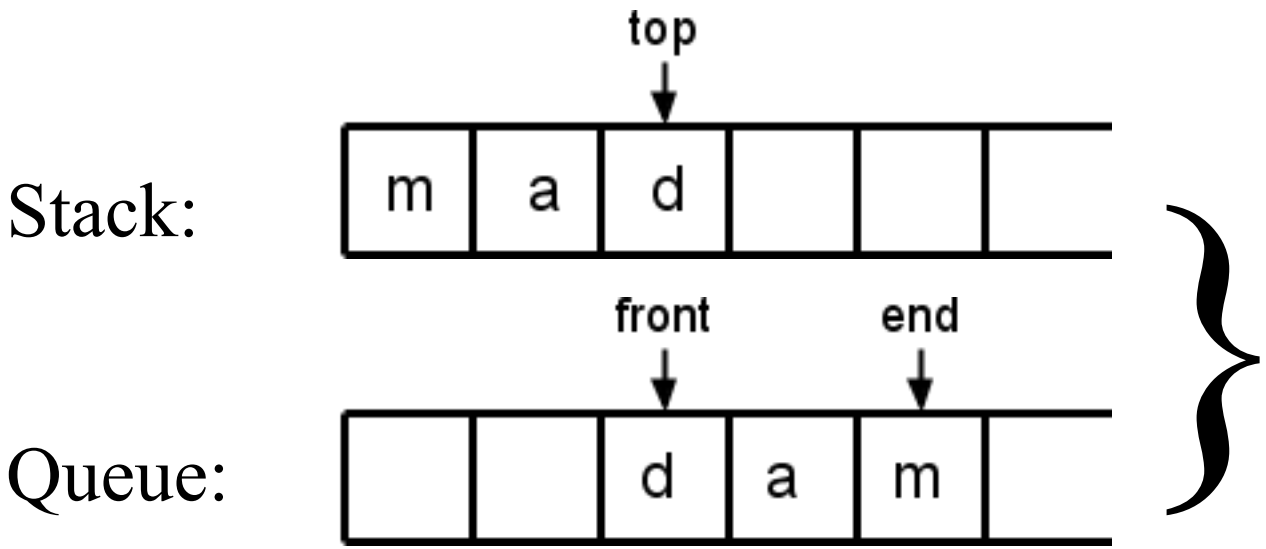
```
bool check_pal() {
    stack s;    queue q;    char ch;

    //read characters and save them
    while( cin >> ch )
    {
        s.push(ch);
        q.insert(ch);
    }

    //check for symmetry
    while( !s.isEmpty() )
    {
        if( s.pop() != q.remove() )
            return false;
    }
    return true;
}
```

Checking for Palindromes

- Assume user enters "madam"

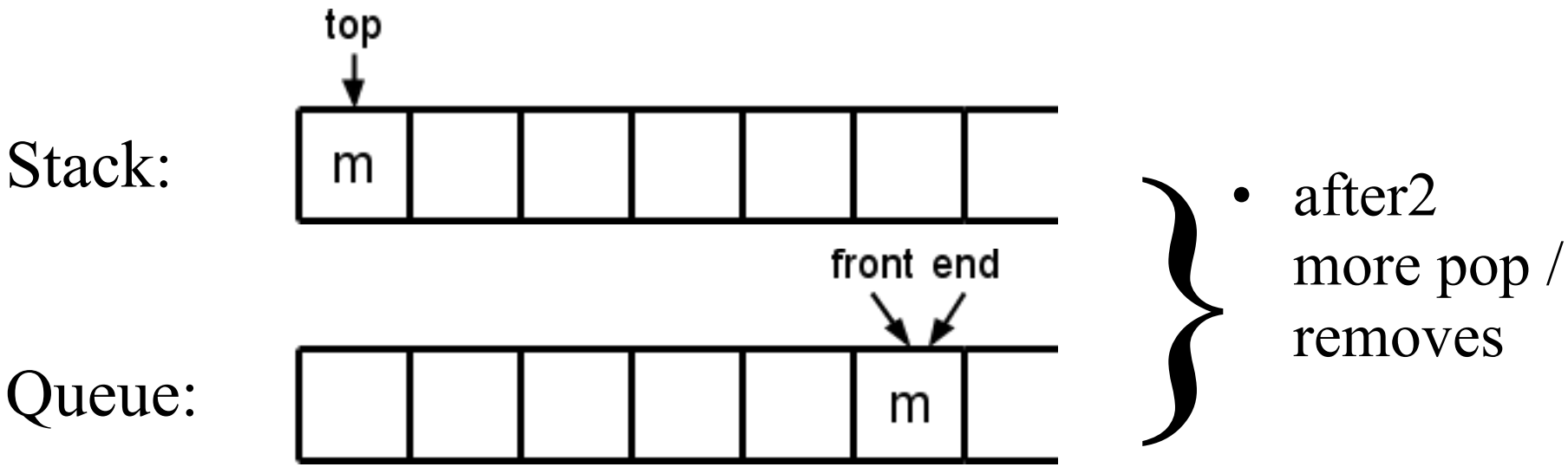


- after all push / inserts



- after 2 pop / removes

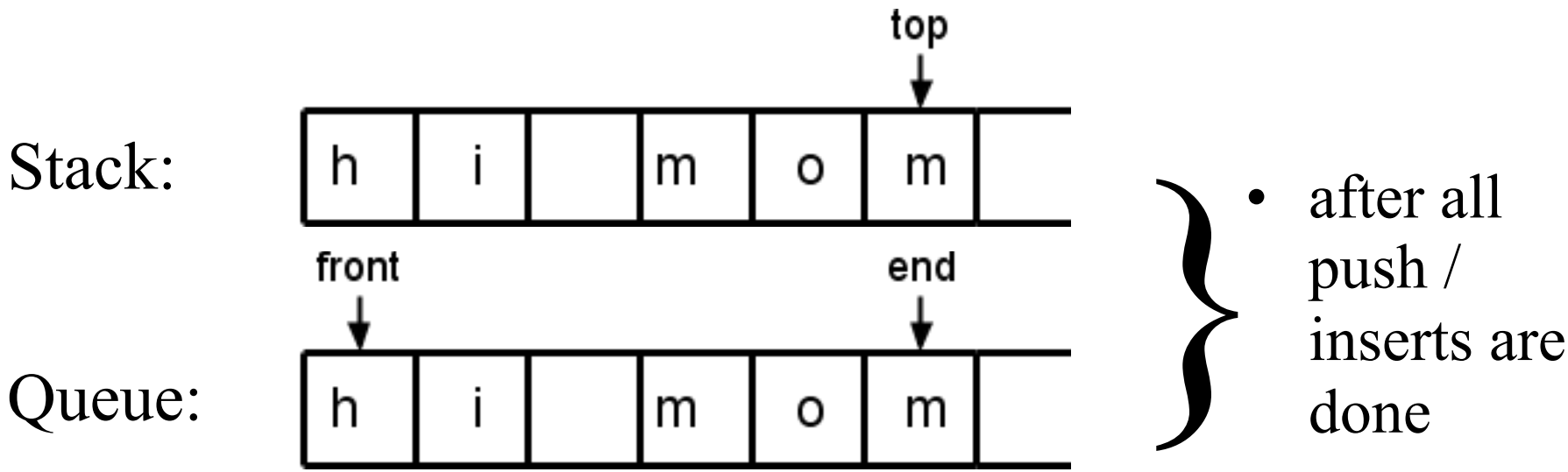
Checking for Palindromes



- All characters match so this is a valid palindrome.

Checking for Palindromes

- Assume user enters "hi mom".



- First s.pop() returns 'm' but first q.remove() returns 'h' so this string is not a palindrome.

Queue Based Flood Fill

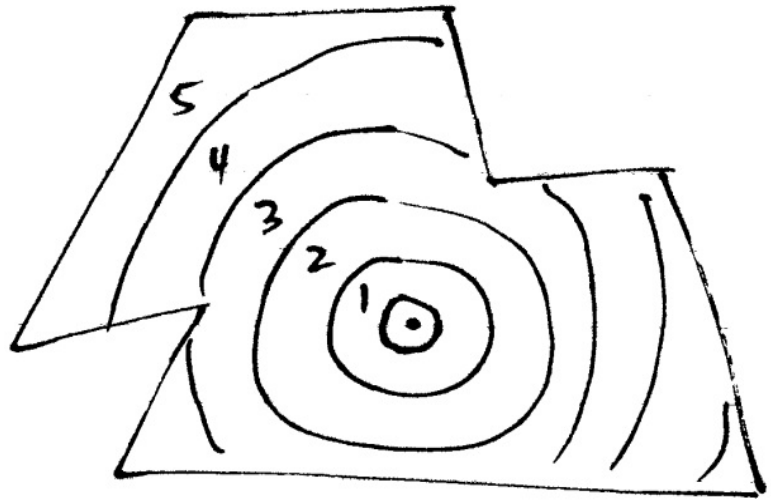
- We can implement flood fill yet again using a queue instead of a stack or recursion.
- When we visit a pixel we insert the 4 neighbors into the queue.
- To decide where to visit next we remove coordinates from the front of the queue.
- This will visit all pixels in the polygon and look more like water flooding if shown in slow motion.
- Because a queue is FIFO, it will not visit pixels in the same order as a stack since it is LIFO.

Queue Based Flood Fill

```
void flood_fill( int x, int y, int color ) {
    //store seed point
    int_queue q;
    q.insert(x);    q.insert(y);

    //remove and process pts on queue
    while( !q.isEmpty() ) {
        //get point
        x=q.remove();    y=q.remove();

        //fill point
        if( pixel[x][y] != color) {
            q.insert(x+1);   q.insert(y);
            q.insert(x-1);    q.insert(y);
            q.insert(x);       q.insert(y+1);
            q.insert(x);       q.insert(y-1);
        }
    }
}
```

Queue Based Flood Fill

- Queue version is almost identical to stack version.



- Sample Polygon.

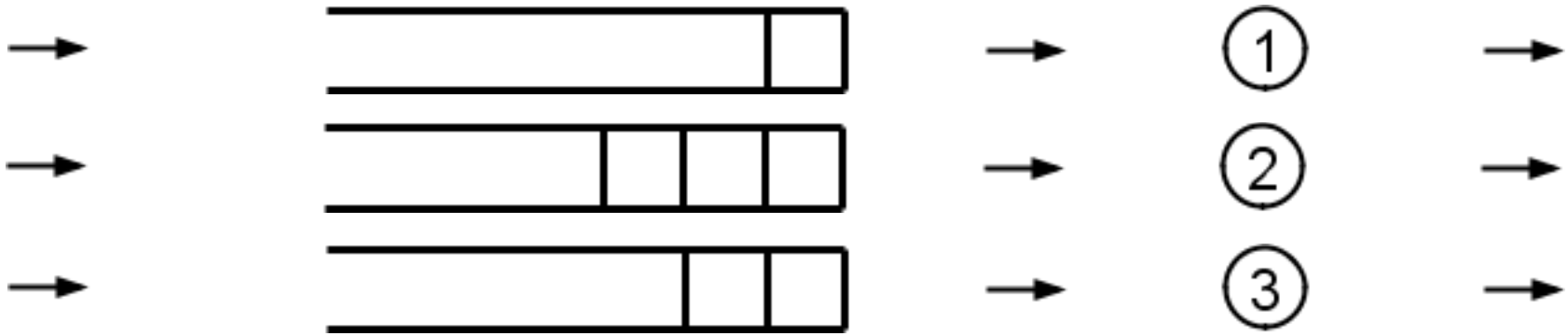
- Order in which pixels are filled is based on distance from the seed point.
- This will look nice when viewed in slow motion.
- This also gives us a way to process pixels based on distance from a given point, which is useful for image analysis and other applications.

Discrete Event Simulations

- Queues are also used to model a wide range of activities where customers wait in line for service.
  - Supermarket checkouts
  - Bank teller windows
  - Gas station pumps
  - Rides at amusement parks
- Store customers in queues as they arrive, and remove customers from queues when they get service.
- Can answer a variety of questions.
  - Average time waiting for service
  - Max / min length of queues
  - Number of clients that "give up”
  - Throughput of servers as group

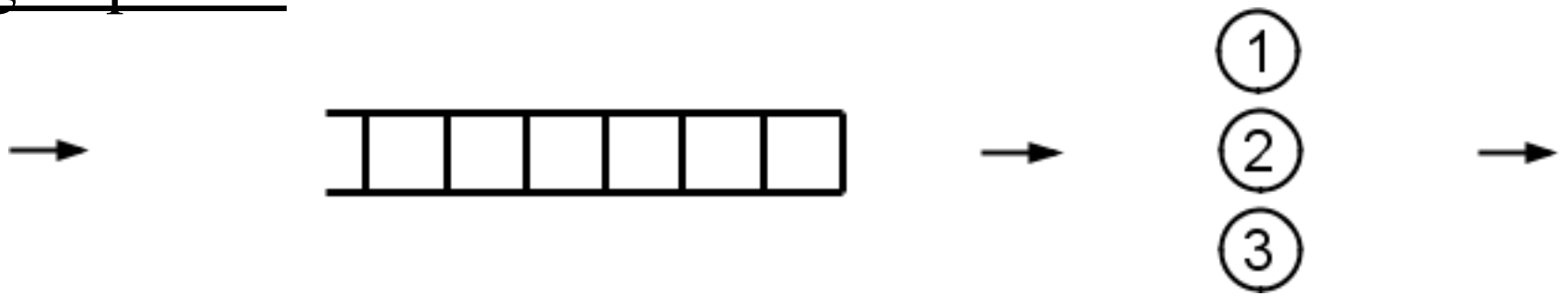
Discrete Event Simulations

multiple queues:



arrive customer queues servers depart

single queues:



arrive customer queues servers depart

- The time between customers is random and the server time is variable.

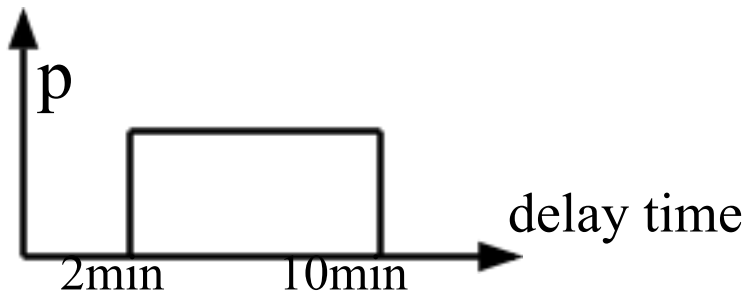
Discrete Event Simulations

- Start virtual clock at time zero.
- initialize all queues to empty.
- loop for all customers (or time limit).
  - get arrival time for customer
  - check all queues / servers to see if anyone is finished yet
  - add customer to shortest queue
  - update virtual clock
- generate simulation statistics based on simulated arrival / departure times.

Discrete Event Simulations

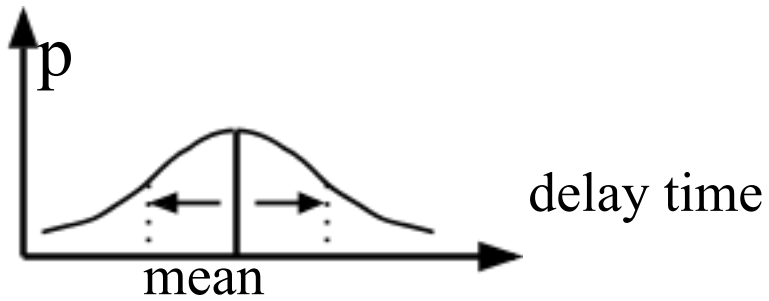
- To get an accurate simulation result we need to know the delays between customer arrivals, and the times needed to provide service.

uniform distribution



- all times between 2min & 10 min are equally likely to occur. (call random)

normal distribution



- delay time modeled by mean, standard distribution.
- can add N uniform random number to simulate normal distribution of numbers.

Other Queue Applications

- On most systems there are several queues running 24x7.

printer queues

- wait for files from users
- will print in FIFO order
- provides shared service to users

communication bufffers

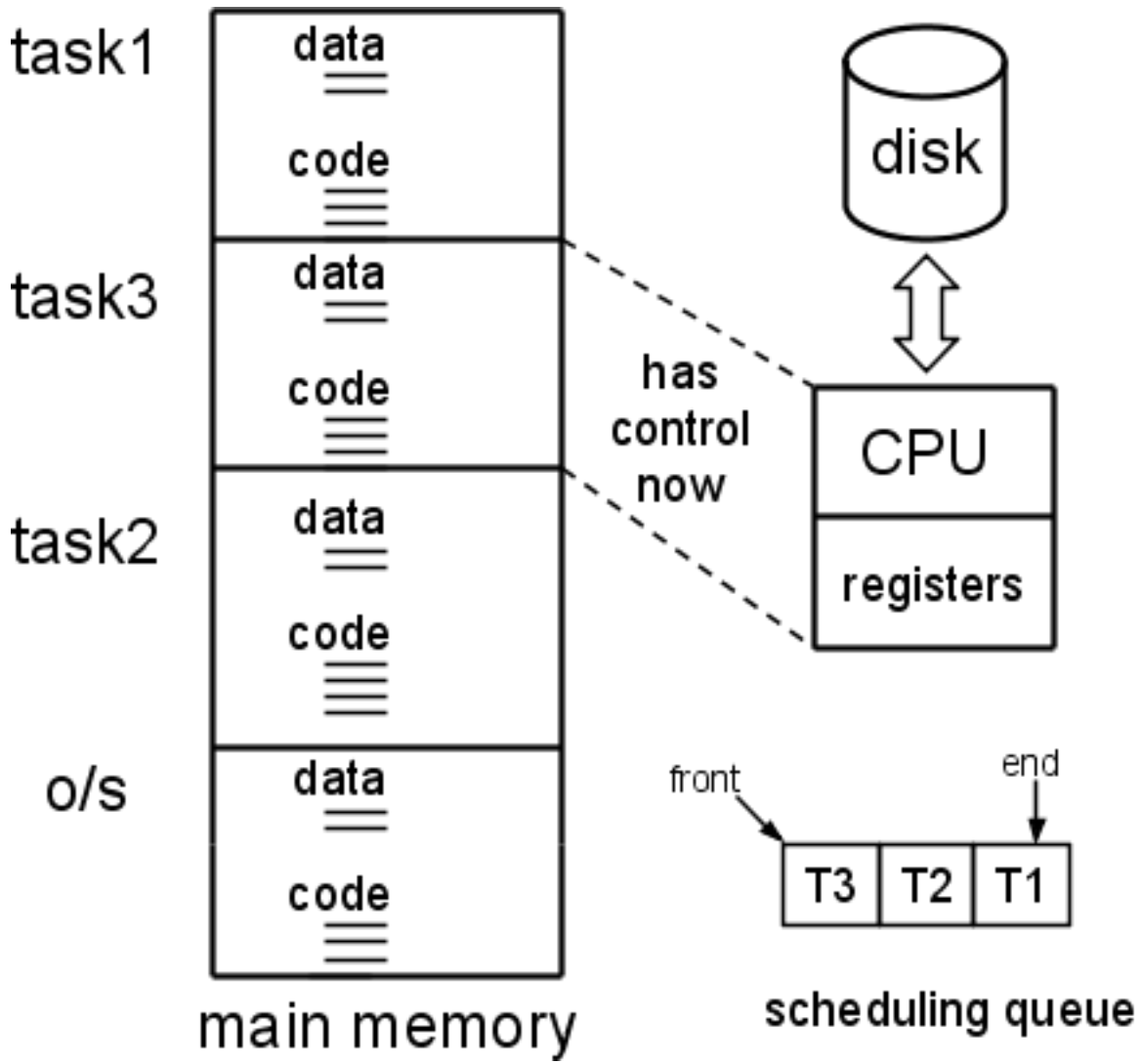
- store data packets in hubs / switches
- process / transmits data in FIFO order
- prevents data loss if there is temporary conjestion

process scheduling

- keeps track of all tasks in a multiprocessing o/s
- will give control of cpu to each task for short time
- when time is up, task goes to end of queue to run again later
- provides fair access to CPU

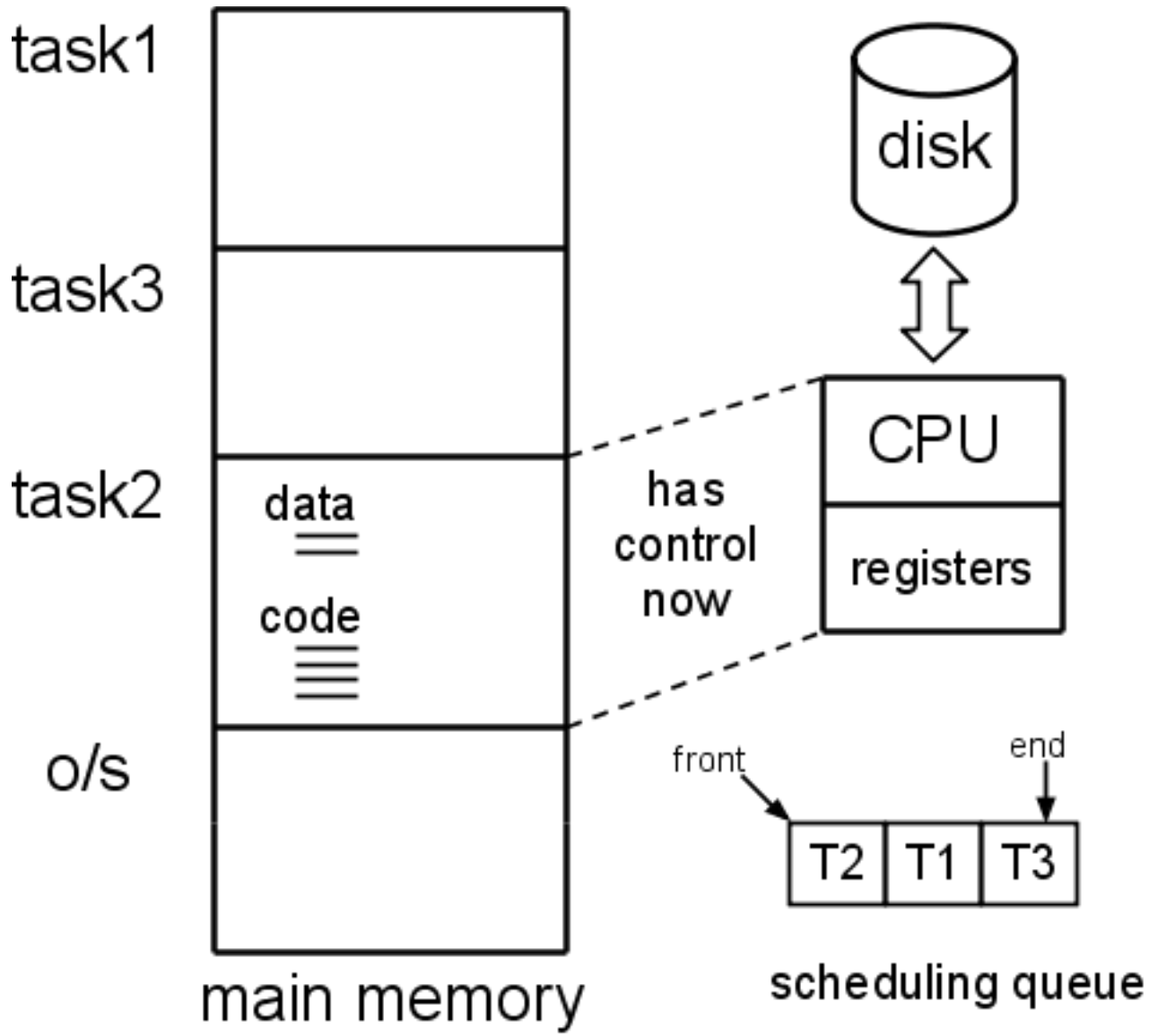
Process Scheduling

- Assume there are 3 tasks in the scheduling queue.
- The code and data for each task are loaded into 3 separate chunks of main memory (and o/s is in the 4th chunk).



Process Scheduling

- When runtime for one task expires, the system saves registers and changes program counter to give control to another task.



Queue Discussion

- The queue ADT is simple to implement using either arrays or linked lists.
- Queues are useful for problems that require "fair" access to a shared resource (printer / cpu).
- Queues are also used extensively in discrete simulations.